1. **What is JPA?**

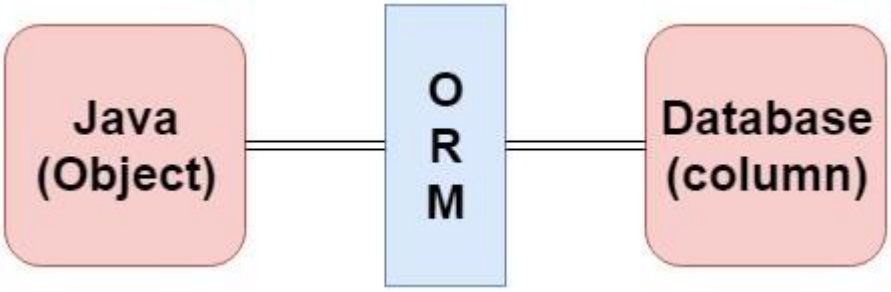
   The Java Persistence API (JPA) is the specification of Java that is used to persist data between Java object and relational database. JPA acts as a bridge between object-oriented domain models and relational database systems.

2. **Does JPA performs the actual task like access, persist and manage data?**

   No, JPA is only a specification. The ORM tools like Hibernate, iBatis, and TopLink implements the JPA specification and perform these type of tasks.

3. **What is the object-relational mapping?**

   The object-relational mapping is a mechanism which is used to develop and maintain a relationship between an object and the relational database by mapping an object state into the database column.



4. **What are the advantages of JPA?**

   - ✓ Reduce the boiler plate of code.
   - ✓ Less Configuration.
   - ✓ Faster development.
   - ✓ Annotations makes developers life easy.

5. **List some ORM frameworks?**

   - ✓ Hibernate
   - ✓ iBATIS
   - ✓ TopLink

6. **What is the JPQL?**

The JPQL (Java Persistence Query Language) is an object-oriented query language which is used to perform database operations on persistent entities. JPQL uses entity object model to operate the SQL queries. Here, the role of JPA is to transform JPQL into SQL.

7. Explain different methods of JPQL?

@Query: @Query can be applied on Methods. It is used to prepare the custom queries. A Query is similar in syntax to SQL, and it's generally used to perform CRUD operations.

```
@Repository
public interface BookRepository extends CrudRepository<Book, Integer> {
  @Query("SELECT b FROM Book b")
List<Book> findAllBooks();
}
```

Native Queries: Native query refers to actual sql queries (referring to actual database objects). These queries are the sql statements which can be directly executed in database. The @Query annotation allows for running native queries by setting the nativeQuery flag to true.

```
public interface UserRepository extends JpaRepository<User, Long> {
@Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1", nativeQuery = true)
User findByEmailAddress(String emailAddress);
}
```

@NamedQuery: We define NamedQuery on the Entity class itself, providing a centralized, quick and easy way to read and find Entity's related queries.

The @NamedQuery annotation contains four elements - two of which are required and two are optional.

- ✓ The two required elements, name and query define the name of the query and the query string itself.
- ✓ The two optional elements, lockMode and hints, provide static replacement for the setLockMode and setHint methods.

```
@Table(name = "users")
@Entity
@NamedQuery(name = "UserEntity.findByUserId", query = "SELECT u FROM UserEntity u WHERE u.id=:userId")
public class UserEntity {
    @Id
    private Long id;
    private String name;
}
```

8. How to update the state of a database?

We can use the @Query annotation to modify the state of the database by also adding the @Modifying annotation to the repository method.

```
@Modifying
@Query("update User u set u.status = :status where u.name = :name")
int updateUserSetStatusForName(@Param("status") Integer status,
  @Param("name") String name);
```

9. What is a JPA entity?

Entities in JPA are nothing but POJOs representing data that can be persisted to the database. An entity represents a table stored in a database. Every instance of an entity represents a row in the table.

10. Explain few annotation you have used on top of Entity?

- ✓ **@Entity:** Is used to define that the class is an Entity class.
- ✓ **@Table:** This annotation is used for specifying the table name which is defined in the database.
- ✓ **@Id:** Id annotation is used for specifying the Id attribute
- ✓ **@GeneratedValue:** This is used when we want to set automatically generated value. GenerationType is the mechanism that is used for generating the value for the specific column.
- ✓ **@Column:** This annotation is used for mapping the columns from the table with the attributes in the class.

11. How to persist the collections entity objects in database?

In JPA, we can persist the object of wrapper classes and String using collections (**List, Set and Map**). We can achieve it by using the **@ElementCollection and @Embeddable**

```
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int e_id;

    @ElementCollection
    private List<Address> address=new ArrayList<Address>();
}
```

The annotation @ElementCollection represents the embedded object.

Now, create a class of embedded object Address.java. The annotation @Embeddable represents the embeddable object.

```
@Embeddable
public class Address {
 private int e_pincode;
 private String e_city;
}
```

12. Explain different types of entity mappings?

One-to-one: The One-To-One mapping represents a single-valued association where an instance of one entity is associated with an instance of another entity.

The one-to-one association can be either unidirectional or bidirectional.

In unidirectional association, the source entity has a relationship field that refers to the target entity and the source entity's table contains the foreign key.

```
Parent/Source:
----------------
@Entity
@Table(name = "instructor")
public class Instructor {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;
private String firstName;

@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "instructor_detail_id")
private InstructorDetail instructorDetail;
}
Child/Destination
------------------------
@Entity
@Table(name = "instructor_detail")
public class InstructorDetail {

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
}
```

In a bidirectional association, each entity (i.e. source and target) has a relationship field that refers to each other and the target entity's table contains the foreign key. The source entity must use the mappedBy attribute to define the bidirectional one-to-one mapping.

```
Parent/Source:
----------------
@Entity
@Table(name = "instructor")
public class Instructor {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
private String firstName;

@OneToOne(cascade = CascadeType.ALL, mappedBy = "instructor", fetch = FetchType.LAZY)
private InstructorDetail instructorDetail;
}
Child/Destination:
----------------------------
@Entity
@Table(name = "instructor_detail")
public class InstructorDetail {

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

@OneToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "instructor_id")
private Instructor instructor;
}
```

One-To-Many & Many-To-One: The One-To-Many mapping comes into the category of collectionvalued association where an entity is associated with a collection of other entities. In this type of association, the instance of one entity can be mapped with any number of instances of another entity.

Unidirectional - In this type of association, only the source entity has a relationship field that refers to the target entity. We can navigate this type of association from one side.

```
Parent/Source:
---------------------
@Entity
@Table(name = "instructor")
public class Instructor {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
@OneToMany(cascade = CascadeType.ALL)
private List < Course > courses = new ArrayList < Course > ();
}

Child/Destination:
------------------------
@Entity
@Table(name = "course")
public class Course {
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 @Column(name = "id")
 private int id;
}
```

Bidirectional - In this type of association, each entity (i.e. source and target) has a relationship field that refers to each other. We can navigate this type of association from both sides.

```
Parent/Source:
------------------------
@Entity
@Table(name = "instructor")
public class Instructor {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;

@OneToMany(mappedBy = "instructor", cascade = { CascadeType.All })
private List<Course> courses;
}
Child/Destination:
---------------------
@Entity
@Table(name = "course")
public class Course {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 @Column(name = "id")
 private int id;

@ManyToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "instructor_id")
private Instructor instructor;
}
```

Many-To-Many: In many-to-many association, the source entity has a field that stores a collection of target entities. The @ManyToMany JPA annotation is used to link the source entity with the target entity

A student can like many courses, and many students can like the same course.

As we know, in RDBMSs we can create relationships with foreign keys. Since both sides should be able to reference the other, we need to create a separate table to hold the foreign keys:

Such a table is called a join table. In a join table, the combination of the foreign keys will be its composite primary key.

Modeling a many-to-many relationship with POJOs is easy. We should include a Collection in both classes, which contains the elements of the others.

After that, we need to mark the class with @Entity and the primary key with @Id to make them proper JPA entities.

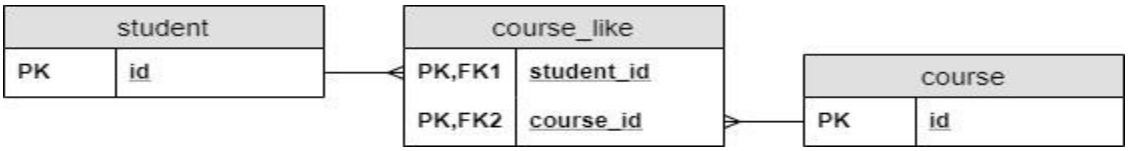
Also, we should configure the relationship type. So, we mark the collections with @ManyToMany annotations

```
Student Class
=================
@Entity
class Student {
@Id
Long id;

@ManyToMany
@JoinTable(
  name = "course_like",
  joinColumns = @JoinColumn(name = "student_id"),
  inverseJoinColumns = @JoinColumn(name = "course_id"))
Set<Course> likedCourses;

}
Course Class:
===============
@Entity
class Course {
@Id
Long id;

@ManyToMany
Set<Student> likes;
}
```



13. Explain about JPA Cascade operation?

In JPA, if any operation is applied on an entity then it will perform on that particular entity only. These operations will not be applicable to the other entities that are related to it.

To establish a dependency between related entities, JPA provides **javax.persistence.CascadeType** enumerated types that define the cascade operations. These cascading operations can be defined with any type of mapping i.e. One-to-One, One-to-Many, Many-to-One, Many-to-Many.

- ✓ **PERSIST:** In this cascade operation, if the parent entity is persisted then all its related entity will also be persisted.
- ✓ **MERGE:** In this cascade operation, if the parent entity is merged then all its related entity will also be merged.
- ✓ **DETACH:** In this cascade operation, if the parent entity is detached then all its related entity will also be detached.
- ✓ **REFRESH**: In this cascade operation, if the parent entity is refreshed then all its related entity will also be refreshed.

- ✓ **REMOVE**: In this cascade operation, if the parent entity is removed then all its related entity will also be removed.
- ✓ **ALL**: In this case, all the above cascade operations can be applied to the entities related to parent entity.

14. What are the different directions of entity mapping?

The direction of a mapping can be either unidirectional or bidirectional. In unidirectional mapping, only one entity can be mapped to another entity, whereas in bidirectional mapping each entity can be mapped or referred to another entity.

15. What is an orphan removal in mappings?

If a target entity in one-to-one or one-to-many mapping is removed from the mapping, then remove operation can be cascaded to the target entity. Such target entities are known as orphans, and the orphanRemoval attribute can be used to specify that orphaned entities should be removed.

16. Explain persistence life cycle of an object?

- ✓ **Transient -** The object is called to be in the transient state when it is just declared by using the new keyword. When an object remains in the transient state, it doesn't contain any identifier(primary key) in the database.
- ✓ **Persistence -** In this state, an object is associated with the session and either saved to a database or retrieved from the database. When an object remains in the persistence state, It contains a row of the database and consists of an identifier value. We can make an object persistent by associating it with hibernate session.
- ✓ **Detached -** The object enters into a detached state when hibernate session is closed. The changes made to the detached objects are not saved to the database.

17. What are the different types of identifier generation?

- ✓ **Automatic Id generation -** In this case, the application doesn't care about the kind of id generation and hand over this task to the provider. If any value is not specified explicitly, the generation type defaults to auto.
- ✓ **Id generation using a table -** The identifiers can also be generated using a database table.
- ✓ **Id generation using a database sequence -** Databases support an internal mechanism for id generation called sequences. To customize the database sequence name, we can use the JPA @SequenceGenerator annotation.

- ✓ **Id generation using a database identity -** In this approach, whenever a row is inserted into the table, a unique identifier is assigned to the identity column that can be used to generate the identifiers for the objects.

18. What is the role of Entity Manager in JPA?

- ✓ The entity manager is used to read, delete and write an entity.
- ✓ An object referenced by an entity is managed by entity manager.

19. What are the constraints on an entity class?

- ✓ The class must have a no-argument constructor.
- ✓ The class can't be final.
- ✓ The class must be annotated with @Entity annotation.
- ✓ The class must implement a Serializable interface if value passes an empty instance as a detached object.

20. Difference between persist and merge?

Persist: It always make a new entity and never updates an entity.

Merge: It inserts or updates an entity in the database.

21. Explain different inheritance strategies in JPA?

Inheritance is a key feature of object-oriented programming language in which a child class can acquire the properties of its parent class. This feature enhances reusability of the code.

The relational database doesn't support the mechanism of inheritance. So, Java Persistence API (JPA) is used to map the key features of inheritance in relational database model.
Below are the types of inheritances in JPA,

- ✓ Single table strategy
- ✓ Joined strategy
- ✓ Table-per-class strategy

Single Table Strategy: The single table strategy is one of the most simplest and efficient way to define the implementation of inheritance. In this approach, instances of the multiple entity classes are stored as attributes in a single table only.

In this example, we will categorize employees into active employees and retired employees. Thus, the subclass **ActiveEmployees** and **RetiredEmployees** inherits the e_id and e_name fields of parent class Employee.

```
@Entity
@Table(name="employee_details")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Employee implements Serializable {
@Id
private int e_id;
private String e_name;
}

//Sub Class
@Entity
public class ActiveEmployee extends Employee {
    private int e_salary;
    private int e_experience;
}

//Sub Class
@Entity
public class RetiredEmployee extends Employee {
    private int e_pension;
}
```

E_ID	DTYPE	E_NAME	E_EXPERIENCE	E_SALARY	E_PENSION
101	ActiveEmployee	Karun	5	10000	*NULL*
102	ActiveEmployee	Rishab	7	12000	*NULL*
103	RetiredEmployee	Ramesh	*NULL*	*NULL*	5000
104	RetiredEmployee	Raj	*NULL*	*NULL*	4000

Joined strategy: In joined strategy, a separate table is generated for every entity class. The attribute of each table is joined with the primary key. It removes the possibility of duplicacy.

Base Class:

```
@Entity
@Table(name="employee_details")
@Inheritance(strategy=InheritanceType.JOINED)
public class Employee implements Serializable {
@Id
private int e_id;
private String e_name;
}

//Sub Class
@Entity
public class ActiveEmployee extends Employee {

    private int e_salary;
    private int e_experience;
}

//Sub Class
@Entity
public class RetiredEmployee extends Employee {
    private int e_pension;
}
```

- Select * from employee_details

E_ID	DTYPE	E_NAME
101	ActiveEmployee	Karun
102	ActiveEmployee	Rishab
103	RetiredEmployee	Ramesh
104	RetiredEmployee	Raj

- Select * from active_employee

E_ID	E_EXPERIENCE	E_SALARY
101 [->]	5	10000
102 [->]	7	12000

- Select * from retired_employee

E_ID	E_PENSION
103 [->]	5000
104 [->]	4000

Table-per-class Strategy: In table-per-class strategy, for each sub entity class a separate table is generated. Unlike joined strategy, no separate table is generated for parent entity class in tableper-class strategy.

```
Base Class:
------------
@Entity
@Table(name="employee_details")
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Employee implements Serializable {
@Id
private int e_id;
private String e_name;
}

//Sub Class
@Entity
public class ActiveEmployee extends Employee {

    private int e_salary;
    private int e_experience;
}

//Sub Class
@Entity
public class RetiredEmployee extends Employee {
    private int e_pension;
}
```

- Select * from active_employee

E_ID	E_EXPERIENCE	E_NAME	E_SALARY
101	5	Karun	10000
102	7	Rishab	12000

- Select * from retired_employee

E_ID	E_NAME	E_PENSION
103	Ramesh	5000
104	Raj	4000

22. Explain different types of JAP Fetch strategies?

FetchType.EAGER: tells Hibernate to get all elements of a relationship when selecting the root entity.

FetchType.LAZY: The persistence provider should load data when it's first accessed, but can be loaded eagerly.

23. How to handle the composite primary key in JPA?

The JPA specification says that all entity identifiers should be Serializable and implement equals and hashCode. So, an Embeddable that is used as a composite identifier must be Serializable and implement equals and hashCode.

```
@Entity(name = "Employee")
@Table(name = "employee")
public class Employee {

    @EmbeddedId
    private EmployeeId id;
    private String name;

    //Getters and Setters
}

@Embeddable
public class EmployeeId implements Serializable {

    @Column(name = "company_id")
    private Long companyId;

    @Column(name = "employee_number")
    private Long employeeNumber;

    //Getters and Setters
}
```

24. In JPA how to map entity directly to DTO?

Return (Data Transfer Object) DTO from the JPA Repository is commonly used in the Spring Data JPA application. DTO is used to return data without exposing the JPA Entity class and sending a limited amount of data is needed for clients via API endpoints. We can return DTO from the JPA Repository using **JPQL with Constructor Expressions, Spring Data JPA Projections** and **Native SQL Queries**.

Approaches to Returning DTOs

- Using JPQL with Constructor Expressions
- Using Spring Data JPA Projections
- Using Native SQL Queries

Using JPQL with Constructor Expressions

In this approach use JPQL to directly instantiate the DTO class objects with their constructors. We need to create a DTO class and then add a custom method annotated with **@Query** annotation in the repository interface.

Step 1: Creating a DTO class

Create a DTO class with a constructor that matches the fields which we need.

```
import lombok.AllArgsConstructor;

import lombok.Data;

@Data

@AllArgsConstructor

public class WebSeriesDTO {

private String name;

private String ott;

}
```

Step 2: Add a custom method in the repository interface

Add a custom method to the repository interface and use JPQL and this method returns DTOs.

```
import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.data.jpa.repository.Query;

public interface WebSeriesRepository extends JpaRepository<WebSeries, Long> {

@Query("SELECT new com.springjava.dto.WebSeriesDTO(w.name, w.ott) FROM WebSeries w")

List<WebSeriesDTO> findAllWebSeriesDTOs();

}
```

- **Constructor Expression:** new com.springjava.dto.WebSeriesDTO(ws.name, ws.ott) instantiates the WebSeriesDTO for each row returned through the query.
- **Field Matching:** Ensure the constructor parameters have matched the query's order and type of fields.

Using Spring Data JPA Projections

In this approach, we can create an interface to return DTO from the repository without making the class's explicit object.

Step 1: Create Projection Interface

Create an interface and define the getter methods for the fields that we want to include in DTO.

```
public interface WebSeriesDTOI {

String getName();

String getOtt();

}
```

Step 2: Add the method to the repository interface

```
import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.data.jpa.repository.Query;

public interface WebSeriesRepository extends JpaRepository<WebSeries, Long> {

List<WebSeriesDTOI> findBy();

}
```

Using Native SQL Queries

In this approach return DTO from JPA Repository using Interface-based DTO and Class-based DTO projection. Using this class-based DTO projection the automatic mapping doesn't work then we need to use @NamedNativeQuery with an @SqlResultSetMapping in the JPA Entity class.

Interface-based DTO with Native Query

In this way, we need to create an Interface with getter methods for fields included in the DTO then add a custom method annotation **@Query** annotation with **nativeQuery = true** attribute in the repository interface.

Step 1: Create an interface

```
public interface WebSeriesDTOI {

String getName();

String getOtt();

}
```

Step 2: Add the custom method to the repository interface

```java
import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.data.jpa.repository.Query;

public interface WebSeriesRepository extends JpaRepository<WebSeries, Long> {

@Query(value="SELECT w.name AS name, w.ott AS ott FROM web_series w",nativeQuery = tru

List<WebSeriesDTO> findAllWebSeriesDTONative();

}
```